

# Faster Radix Sort via Virtual Memory and Write-Combining

Jan Wassenberg  
Fraunhofer IOSB

jan.wassenberg@iosb.fraunhofer.de

Peter Sanders  
KIT

September 7, 2010

## Abstract

Sorting algorithms are the deciding factor for the performance of common operations such as removal of duplicates or database sort-merge joins. This work focuses on 32-bit integer keys, optionally paired with a 32-bit value. We present a fast radix sorting algorithm that builds upon a microarchitecture-aware variant of counting sort. Taking advantage of virtual memory and making use of write-combining yields a per-pass throughput corresponding to at least 88 % of the system's peak memory bandwidth. Our implementation outperforms Intel's recently published radix sort by a factor of 1.5. It also compares favorably to the reported performance of an algorithm for Fermi GPUs when data-transfer overhead is included. These results indicate that scalar, bandwidth-sensitive sorting algorithms remain competitive on current architectures. Various other memory-intensive applications can benefit from the techniques described herein.

applications can often replace large records with a pointer or index [1]. The radix sort algorithm is commonly used in such cases due to its  $O(N)$  complexity. In this preliminary report, we show a 1.5-fold performance increase over results recently published by Intel [2].

The remaining sections are organized in a bottom-up fashion, with Section 2 dedicated to the basic realities of current and future microarchitectures that affect memory-intensive programs and motivate our approach. We build upon this foundation in Section 3, showing how to speed up counting sort by taking advantage of virtual memory and write-combining. Section 4 applies this technique towards our main contribution, a novel variant of radix sort. The performance of our implementation is evaluated in Section 5. Bandwidth measurements indicate the per-pass throughput is nearly optimal for the given hardware. Its two CPUs outperform a Fermi GPU when accounting for data-transfer overhead.

## 1 Introduction

Sorting is a fundamental operation that is a time-critical component of various applications such as databases and search engines. The well-known lower bound of  $\Omega(n \cdot \log n)$  for comparison-based algorithms no longer applies when special properties of the keys can be assumed. In this work, we focus on 32-bit integer keys, optionally paired with a 32-bit value (though larger sizes are possible). This simplifies the implementation without loss of generality, since

## 2 Software Write-Combining

We begin with a description of basic microarchitectural realities that are likely to have a serious impact on applications with numerous memory accesses, and show how to avoid performance penalties by means of Software Write-Combining. These topics are not new, but we believe they are often not adequately addressed.

The first problem arises when writing items to multiple streams. An ideal cache with at least as many

lines could exploit the writes’ spatial locality and entirely avoid noncompulsory misses. However, perfect hit rates are not achievable in practice due to limited ways of associativity  $a$  [3]. Since only  $a$  lines can be mapped to a cache set, any further allocations from that set result in the eviction of one of the previous lines. If possible, applications should avoid writing to many different streams. Otherwise, the various write positions should map to different sets to avoid thrashing and conflict misses. For current L1 caches with  $a = 8$  ways, size  $C = 32$  KiB and lines of  $B = 64$  bytes, there are  $S = \frac{C}{a \cdot B} = 64$  sets, and bits  $\lceil \lg B, \lg B + \lg S \rceil$  of the destination addresses should differ (e.g. by ensuring the write positions are not a multiple of  $S \cdot B = 4$  KiB apart).

A second issue is provoked by a large number of write-only accesses. Even if an entire cache line is to be written, the previous destination memory must first be read into the cache. While the corresponding latency may be partially hidden via prefetching, the cache line allocations remain problematic due to capacity constraints and eviction policy. Instead of displacing write-only lines that are not accessed after having been filled, the widespread (pseudo-)Least-Recently-Used strategy displaces previously cached data due to their older timestamp. An attempt to avoid these evictions by explicitly invalidating cache lines (e.g. with the IA-32 CLFLUSH instruction) did not yield meaningful improvements. Instead, applications should avoid ‘cache pollution’ by writing directly to memory via non-temporal streaming stores.

This leads directly to the next concern: single memory accesses involve significant bus overhead. The architecture therefore combines neighboring non-temporal writes into a single burst transfer. However, currently microarchitectures only provide four to ten write-combine (WC) buffers [4]. Non-temporal writes to multiple streams may force these buffers to be flushed to memory via ‘partial writes’ before they are full. The application can prevent this by making use of Software Write-Combining [5]. The data to be written is first placed into temporary buffers, which almost certainly reside in the cache because they are frequently accessed. When full, a buffer is copied to the actual destination via consecutive non-temporal writes, which are guaranteed to be combined into a

single burst transfer.

This scheme avoids reading the destination memory, which may incur relatively expensive Read-For-Ownership transactions and would only pollute the cache. It works around the limited number of WC buffers by using L1 cache lines for that purpose. Interestingly, this is tantamount to direct software control of the transparently managed cache.

We recommend the use of such Software Write-Combining whenever a core’s active write destinations outnumber its write-combine buffers. Fortunately, this can be done at a fairly high level, since only the buffer copying requires special vector loads and non-temporal stores (which are best expressed by the SSE2 intrinsics built into the major compilers).

### 3 Virtual-Memory Counting Sort

We now review Counting Sort and describe an improved variant that makes use of virtual memory and write-combining.

The naïve algorithm first generates a histogram of the  $N$  keys. After computing the prefix sum to yield the starting output location for each key, each value is written at its key’s output position, which is subsequently incremented.

Our first optimization goal is to avoid the initial counting pass. We could instead insert each value into a per-key container, e.g. a list of buckets. However, this incurs some overhead for checking whether the current bucket is full. A large array of  $M$  pre-allocated buckets is more efficient, because items can simply be written to the next free position (c.f. Algorithm 1, introduced in [6]). This algorithm only

#### Algorithm 1: Single-pass counting sort

```

storage := ReserveAddressSpace( $N \cdot M$ );
for  $i := 0$  to  $M$  do next[i] :=  $i \cdot N$ ;
foreach key,value do
    storage[next[key]] := value;
    next[key] := next[key] + 1;

```

writes and reads each item once, a feat that comes at

the price of  $N \cdot M$  space. While this appears problematic in the Random-Access-Machine model, it is easily handled by 64-bit CPUs with paged virtual memory. Physical memory is only mapped to pages when they are first accessed,<sup>1</sup> thus reducing the actual memory requirements to  $O(N + M \cdot \text{pageSize})$ . The remainder of the initial allocation only occupies address space, of which multiple terabytes are available on 64-bit systems.

Having avoided the initial counting pass, we now show how to efficiently write values to **storage** using the write-combining technique described in Section 2. Our implementation initializes the next pointers to consecutive, naturally aligned, cache-line-sized buffers. A buffer is full when its (post-incremented) position is evenly divisible by its size. When that happens, an unrolled loop of non-temporal writes copies the buffer to its key’s current output position within **storage**. These output positions are also stored in an array of pointers.

## 4 Radix Sort

After a brief review of radix sorting, we introduce a new variant based on the virtual-memory counting sort described in Section 3.

A radix sort successively examines  $D$ -bit ‘digits’ of the  $K$ -bit keys. They are characterized by the order in which digits are processed: starting at the Least Significant Digit (LSD), or Most Significant Digit (MSD).

An MSD radix sort partitions the items according to the current digit, then recursively sorts the resulting buckets. While it no longer needs to move items whose previously seen key digits are unique, this is not especially helpful when the number of passes  $K/D$  is small. In fact, the overhead of managing numerous (nearly empty) buckets makes MSD radix sort less suited for relatively small  $N$ .

By contrast, each iteration of the LSD variant par-

<sup>1</sup>Accesses to non-present pages result in a page fault exception. The application receives such events via signals (POSIX) or Vectored Exception Handling (Microsoft Windows) and reacts by committing memory, after which the faulting instruction is repeated.

### Algorithm 2: Parallel Radix Sort

```

parallel foreach item do
   $d := \text{Digit}(\text{item}, 3);$ 
   $\text{buckets3}[d] := \text{buckets3}[d] \cup \{\text{item}\};$ 
Barrier;
foreach  $i \in [0, 2^D)$  do
   $\text{bucketSizes}[i] := \sum_{\text{PE}} |\text{buckets3}[i]|;$ 
   $\text{outputIndices} := \text{PrefixSum}(\text{bucketSizes});$ 
parallel foreach  $\text{bucket3} \in \text{buckets3}$  do
  foreach item  $\in \text{bucket3} \forall \text{PE}$  do
     $d := \text{Digit}(\text{item}, 0);$ 
     $\text{buckets0}[d] := \text{buckets0}[d] \cup \{\text{item}\};$ 
    foreach  $\text{bucket0} \in \text{buckets0}$  do
      foreach item  $\in \text{bucket0}$  do
         $d := \text{Digit}(\text{item}, 1);$ 
         $\text{buckets1}[d] := \text{buckets1}[d] \cup \{\text{item}\};$ 
         $d := \text{Digit}(\text{item}, 2);$ 
         $\text{histogram2}[d] := \text{histogram2}[d] + 1;$ 
      foreach  $\text{bucket1} \in \text{buckets1}$  do
        foreach item  $\in \text{bucket1}$  do
           $d := \text{Digit}(\text{item}, 2);$ 
           $i := \text{outputIndices}[d] + \text{histogram2}[d];$ 
           $\text{histogram2}[d] := \text{histogram2}[d] + 1;$ 
           $\text{output}[i] := \text{item};$ 

```

titions *all* items into buckets by the current key digit. Since buckets are not recursively split, their sizes are nearly equal (under the assumption of a uniform key distribution) and the sort is stable (preserving the original relative order of values with equal keys). However, this comes at the cost of more copying.

To reduce this overhead and also parallel communication, we make use of “reverse sorting” [7], in which one or more MSD passes partition the data into buckets, which are then locally sorted via LSD. This turns out to be even more advantageous for Non-Uniform Memory Access (NUMA) systems because each processor is responsible for writing a contiguous range of outputs, thus ensuring the OS allocates those pages from the processor’s NUMA node [8].

Let us now examine the pseudocode of the radix sort (Algorithm 2), choosing  $K = 32$  for brevity and  $D = 8$  to allow extracting key digits without masking. Each Processing Element (PE) first uses count-

ing sort to partition its items into local buckets by the MSD (digit = 3). Note that items consist of a key and value, which are adjacent in memory (ideally within a native 64-bit word, but larger combinations are possible in our implementation via larger user-defined types). After all are finished, the output index of the first item of a given MSD is computed via prefix sum. Each PE is assigned a range of MSD values, sorting the buckets from all PEs for each value. Note that skewed MSD distributions cause load imbalance, which can be resolved by one or more additional recursive MSD passes (left for future work). The local sort entails  $K/D - 1$  iterations in LSD order. The first copies the other PE’s buckets into local memory. Pass  $K/D - 1$  also computes the histogram of the final digit. This allows writing directly to the output positions in the final pass. Note that three sets of buckets are required, which makes heavy use of virtual memory ( $3 \cdot 2^D \cdot |PE| = 6144$  times the input size). While 64-bit Linux grants each process 128 TiB address space, Windows limits this to 8 TiB, which means only about 1 GiB of inputs can be sorted. This restriction can be lifted when the key distribution is known and each bucket does not need to pre-allocate storage for all  $N$  items.

We briefly discuss additional system-specific considerations. The radix  $2^D$  was motivated by easy access to each digit, but is also limited by the cache and TLB size. Because of the many required TLB entries, we map the buckets with small pages, for which the Intel i7 microarchitecture has 512 second-level TLB entries. To increase TLB coverage, we use large pages for the inputs. The working set consists of  $2^D$  buffers, buffer pointers, output positions, and 32-bit histogram counters. This fits in a 32 KiB L1 data cache if the software write-combine buffers are limited to a single 64-byte cache line. To avoid associativity and aliasing conflicts, these arrays are contiguous in memory. Interestingly, these optimizations do not detract from the readability of the source code. Knowledge of the microarchitecture can also be applied towards middle-level languages and enables principled design decisions.

## 5 Performance Evaluation

We characterize the performance of our sorting implementation by its throughput, defined as  $\frac{N}{t_1 - t_0}$ , where  $N = 64$  Mi and  $t_0$  and  $t_1$  are the earliest and latest start and finish times reported by any thread. The test platform consists of dual W5580 CPUs (3.2 GHz, 48 GiB DDR3-1066 memory) running Windows XP x64. Our implementation is compiled with ICC 11.1.082 /Ox /Og /Oi /Ot /Qipo /GA /EHsc /MD /GS- /fp:fast=2 /GR- /Qopenmp /QaxSSE4.2 /Quse-intel-optimized-headers. For uniformly distributed 32-bit keys generated by the WELL512 algorithm [9] and no associated values, the basic algorithm (‘VM only’) reaches a throughput of 334 M/s, as shown in the second column of Table 1. When write-combining is enabled (‘VM+WC’), performance nearly doubles to 621 M/s.

Intel has reported 240 M/s for the same task and a single but identical CPU [2]. For a fair comparison with our dual-CPU system, we double the given throughput, which assumes their algorithm is NUMA-aware, scales perfectly and is not running at a lower memory clock (since DDR3-1066 is at the lower end of currently available frequencies). We must also divide by the given speedup of 1.2 due to hyperthreads, since those are disabled on our machine. This (‘Intel x2’) yields 400 M/s; the proposed algorithm is therefore more than 1.5 times as fast.

A separate publication has also presented results [10] for the Many Integrated Cores architecture. The Knights Ferry processor provides 32 cores, each with 4 threads and 16-wide SIMD. The simulation (‘KNF MIC’) shows a throughput of 560 M/s. Our scalar implementation is currently 1.1 times as fast when running on 8 cores.

Recently, a throughput of 1005 M/s was reported on a GTX 480 (Fermi) GPU [11]. However, this excludes driver and data-transfer overhead. For applications in which the data is generated and consumed by the CPU, we must include at least the time required to read and write data over the PCIe 2.0 bus. Assuming the peak per-direction bandwidth of 8 GB/s is reached, the aggregate throughput (‘GPU+PCIe’) is 501 M/s. Our implementation, running on two CPUs, therefore outperforms this al-

gorithm on the current top-of-the-line GPU by a factor of 1.24 despite lower transistor counts ( $2 \cdot 731$  M vs. 3000 M) and thermal design power ( $2 \cdot 130$  W vs. 275..300 W).

Algorithm	K=32,V=0	K=32, V=32
VM only	334	203
Intel x2	400	307
GPU+PCIe	501	303
KNF MIC	560	(unknown)
VM+WC	621	430

Table 1: Throughputs [million items per second] for 32-bit keys and optional 32-bit values.

Similar measurements and extrapolations for the case of 32-bit keys associated with  $V = 32$ -bit values are given in the third column of Table 1. Since the slowdown is less than a factor of two, the implementations are at least partially limited by computation and not bandwidth. Intel’s algorithm is more efficient in this regard, with only a 1.3-fold decrease vs. our factor of 1.4. The additional data transfers over PCIe render the GPU algorithm uncompetitive.

To better characterize performance, we measured the exact traffic at each socket’s memory controller. Since this information is not available from current profilers such as VTune (which use per-core performance counters), we have developed a small kernel-mode driver to provide access to the model-specific performance counters in the Intel i7 uncore<sup>2</sup>. Uncached writes constitute the bulk of the write combiners’ memory traffic and are therefore of particular interest. They are apparently reported as Invalid-To-Exclusive transitions and can thus be counted as the total number of *reads* minus ‘normal’ reads [12]. We find that 2041 MiB are written, which corresponds to  $64 \text{ Mi items} \cdot 8 \text{ bytes per item} \cdot 4 \text{ passes}$  (slightly less because our final pass cannot use non-temporal writes when the output position is not aligned). Surprisingly, 2272 MiB are read. The cause of the additional 10 % is unknown and will be investigated in future work. However, we can provide a conservative

<sup>2</sup>The part of the socket not associated with a particular core.

estimate of the bandwidth utilization. Given the pure read and write bandwidths (38687 MB/s and 28200 MB/s) measured by RightMark [13], the minimum time required to read and write the items’ 2048 MiB is 132 ms, which is 88 % of the total measured time. Since this calculation does not include write-to-read turnaround [14, p. 486], there is even less room for improvement than indicated.

## 6 Conclusion

We have introduced improvements to counting sort and a novel variant of radix sort for integer key/value pairs. Bandwidth measurements indicate our algorithm’s throughput is within 12 % of the theoretical optimum for the given hardware. It outperforms the recently published results of Intel’s radix sort by a factor of 1.5 and also outpaces a Fermi GPU when data transfer overhead is included. These results indicate that scalar, bandwidth-sensitive sorting algorithms still have their place on current architectures. We believe the general software write-combining technique can provide similar speedups for other memory-intensive applications.

## References

- [1] P. Bohannon, P. McIlroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *SIGMOD Conference*, pages 163–174, 2001.
- [2] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In A. Elmagarmid and D. Agrawal, editors, *SIGMOD Conference*, pages 351–362. ACM, 2010.
- [3] Mehlhorn and Sanders. Scanning multiple sequences via cache memory. *Algorithmica*, 35, 2003.
- [4] Intel. *Intel Architecture Software Developer Manual*, 2010. System Programming Guide.

- [5] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, November 2009.
- [6] J. Wassenberg, W. Middelmann, and P. Sanders. An Efficient Parallel Algorithm for Graph-Based Image Segmentation. <http://algo2.iti.uni-karlsruhe.de/wassenberg/wassenberg09parallelSegmentation.pdf>, June 2009.
- [7] D. Jimenez-Gonzalez, J. Navarro, and J. Larriba-Pey. Fast parallel in-memory 64-bit sorting. In *Proceedings of the 2001 International Conference on Supercomputing (15th ICS'01)*, pages 114–122, Sorrento, Napoli, Italy, June 2001. ACM.
- [8] D. an Mey and C. Terboven. Affinity Matters! OpenMP on Multicore and ccNUMA Architectures. In *Parallel Computing: Architectures, Algorithms and Applications*, volume 15, February 2008.
- [9] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software*, 32, 2006.
- [10] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey. Fast sort on CPUs, GPUs and intel MIC architectures. Technical report, Intel, 2010.
- [11] D. Merrill and A. Grimshaw. Revisiting sorting for GPGPU stream architectures. Technical Report 3, University of Virginia, February 2010.
- [12] D. Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. Intel.
- [13] D. Besedin. RightMark Memory Analyzer. <http://cpu.rightmark.org>. Accessed 2009-01-09.
- [14] B Jacob S. Ng D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2007.